**Package Routing Solution**

**Tyler Bradshaw**

**Data Structures and Algorithms II**

**A. Nearest Neighbor Algorithm**

The Nearest Neighbor Algorithm is a greedy algorithm used primarily for solving the traveling salesman problem (TSP) and similar routing problems. The main idea behind the algorithm is to start at a particular point and then repeatedly visit the nearest unvisited point until all points have been visited.

**B. Self-adjusting data structure: Hash Table**

A hash table is a data structure that enables the mapping of keys to their corresponding values. It employs a hash function to determine an index, directing it to a specific slot in an array where the associated value resides.

In the context of package data, each package can have various attributes (e.g., delivery address, deadline, weight, etc.). A hash table can store these attributes as values associated with a unique key (e.g., package ID).

When a package's data needs to be retrieved, the hash function processes the key and directs the algorithm to the appropriate slot where the package's data can be found. This allows for rapid retrieval based on the key without regard for the data size.

**C. Program Overview**

**1.**

**Stated Problem:**

This project aims to develop a Python algorithm to manage and deliver packages efficiently. Given a list of packages with various attributes (e.g., delivery address, deadline, weight, etc.), the algorithm should determine an efficient route for delivery using the nearest-neighbor approach. The package data will be stored in a hash table for rapid retrieval and updates.

**Algorithm Overview:**

The delivery management algorithm is constructed in the following manner:

**Nearest Neighbor with Hash Table** **PackageData[1 ... n]**, **n = length[PackageData]**

**Hash Table Operations**:

1. **Hash-Insert**: Inserts a package into the hash table.
   * **Hash-Insert ( HashTable, key, package )**: **HashTable[key] = package**
2. **Hash-Retrieve**: Retrieves a package's data using its key.
   * **Hash-Retrieve ( HashTable, key )**: **return HashTable[key]**
3. **Hash-Remove**: Removes a package from the hash table using its key.
   * **Hash-Remove ( HashTable, key )**: **delete HashTable[key]**

**Running Times**: **Hash-Insert**: *O(1)* on average **Hash-Retrieve**: *O(1)* on average **Hash-Remove**: *O(1)* on average

**Nearest Neighbor Algorithm**: **NearestNeighborAlgorithm (PackageData, CurrentLocation)**

1. **Initialize**:
   * **HashTable = {}**
   * **CurrentLocation = HubLocation**
2. **Insert Packages**:
   * **for each package in PackageData do**
     + **key = package.ID**
     + **Hash-Insert ( HashTable, key, package )**
3. **Delivery Process**:
   * **while HashTable is not empty**
     + **NearestPackage = FindNearestPackage (HashTable, CurrentLocation)**
     + **Deliver (NearestPackage)**
     + **Hash-Remove ( HashTable, NearestPackage.ID )**
     + **CurrentLocation = NearestPackage.location**
4. **Find Nearest Package**:
   * **FindNearestPackage (HashTable, CurrentLocation)**
   * **NearestDistance = infinity**
   * **for each package in HashTable do**
     + **Distance = CalculateDistance (CurrentLocation, package.location)**
     + **if Distance < NearestDistance then**
       - **NearestPackage = package**
       - **NearestDistance = Distance**
   * **return NearestPackage**
5. **Completion**:
   * **ReturnToHub (CurrentLocation)**
   * **CurrentLocation = HubLocation**

**Running Time**: The running time for the Nearest Neighbor Algorithm is *O(n^2)* due to the nearest neighbor search for *n* packages.

1. **Programming Environment**:
   * **Software**: The proposed algorithm will be created with PyCharm within the Python 3.9.7 environment, augmented with the Tkinter module for GUI functionalities.
   * **Hardware**:

Processor Intel(R) Core (TM) i9-10850K CPU @ 3.60GHz 3.60 GHz

Installed RAM 32.0 GB 3600 MHZ

Graphics AMD 6900XT 16 GB GDDR6

System type 64-bit operating system, x64-based processor

1. **Space-time Complexity**:

Analyzing the space-time complexity reveals distinct behaviors of our primary components. The Nearest Neighbor Algorithm exhibits an *O(n^2)* time complexity, attributed to its iterative search for the nearest package among *n* packages. In contrast, hash table operations, encompassing insertions and retrievals, consistently offer an average time complexity of *O(1*). Hence, the overarching time complexity of the program is dominated by the behavior of the nearest neighbor search *O(n^2).*

1. **Scalability**:

The hash table inherently offers efficient scaling with increasing data, ensuring consistent retrieval times regardless of the number of packages. However, the Nearest Neighbor Algorithm may face challenges with extensive datasets due to its *O(n^2)* complexity. For large datasets, algorithms such as Divide and Conquer or dynamic programming may offer more efficient routing solutions.

1. **Efficiency and Maintenance**:

The hash table's constant time complexity for data operations enhances the architecture's efficiency and maintainability. Python's standard libraries and modular code design ensure streamlined maintenance and adaptability.

1. **Strengths and Weaknesses of the Hash Table**:

While the hash table offers advantages like constant average time complexity and memory efficiency, there are challenges. A robust hash function is imperative to minimize collisions. Moreover, performance can be impacted if the table's load becomes too large, necessitating efficient rehashing strategies.

1. **Key Choice for Efficient Delivery Management**:

The package ID is the most suitable key for efficient delivery management. This is because it is unique for each package, and it does not change over time. Using other attributes like delivery address or deadline might result in collisions in the hash table, especially if multiple packages have the same address or deadline.